Derradji Fahd
Geisenberger Baptiste
TD5

Compte rendu du projet :

Tout au long du projet, nous avons travaillé côte à côte, soit à dauphine dans les salles informatiques en B0…, soit chacun sur notre ordinateur par teams. Ce travail d'équipe nous a permis de constamment confronter nos idées, et de réfléchir à deux pour tenter de choisir la meilleure méthode afin d'avancer efficacement. Tout au long du projet, nous avons réfléchi à :

- Optimiser la mémoire.
- Optimiser les complexités des algorithmes.
- Essayer de fournir un code propre (même si cet objectif a un peu été mis sur le côté pour privilégier les deux premiers.

Petite aparté, nous avons choisi des valeurs maximales pour un bon nombre de chaines de caractères de 100. En effet, la taille des lignes de codes dépendait principalement de la taille de l'étiquette, qui n'avait aucune raison d'être bien plus grande qu'un ou deux mots. Le nom du fichier non plus n'avait aucune raison d'excéder 100.

Traduction :

En ouvrant le sujet, nous avons tout de suite été excité de découvrir un sujet très libre, qui donnait beaucoup de la place à la réflexion. En effet, on a vite compris qu'il valait mieux de ne pas commencer à programmer trop hâtivement. Donc, avant de commencer à coder, on a réfléchi un certain temps à quel plan de travail adopter. On s'est décidé de suivre le plan suivant :

- Parcourir le fichier une première fois pour stocker toutes les étiquettes et leurs informations.
- Reparcourir notre fichier une deuxième fois en traduisant chaque ligne, et en insérant celle-ci directement dans notre nouveau fichier hexa.pgm.

La première difficulté qui semblait se dégager de ce sujet était de bien savoir s'approprier les étiquettes. Il fallait dans un premier temps parcourir notre fichier, pour garder comme informations le nom des différentes étiquettes, mais aussi leur place dans le programme. Nos deux premières idées, à savoir la création d'un tableau à double entrées ou de deux tableaux à une entrée ont été mise de côtés, et on a privilégié la création d'une structure etiq, qui permettait un code plus lisible. Afin d'optimiser notre programme, nous avons aussi décider de distinguer deux cas : le premier pour un programme sans aucune étiquette, le deuxième pour un programme avec des étiquettes. Il suffisait de voir si la première ligne commençait par un espace ou une tabulation, ou bien de voir si elle contenait un ':'. Ceci nous permettait de ne pas avoir à parcourir le programme s'il ne possédait pas d'étiquette. Cette décision s'est avérée être très mauvaise, car nous avons réalisé après coup qu'une ligne qui n'était pas précédé par une étiquette pouvait commencer par un nombre indéterminé d'espaces : il était par conséquent impossible de distinguer en une seule ligne si notre code contenait ou non des étiquettes. On avait donc codé deux cas différents, et deux fonctions « traduction » différentes, alors qu'elles ne servaient en réalité à rien : il fallait parcourir tout le fichier une première fois dans tous les cas. Originellement, on cherchait à éviter ce premier parcours s'il n'était pas nécessaire, mais on a réalisé que celui-ci n'alourdissait pas vraiment notre programme par rapport aux nombreux traitements de chaque ligne qu'on faisait par la suite.
Nous avons d'abord eu du mal à bien initialiser notre tableau d'etiq, à cause de certains oublis sur les notions de pointeurs et de pointeurs sur pointeurs, qui ne simplifiaient pas la tâche. En effet, on ne savait plus s'il était possible de créer un pointeur sur etiq pour toutes les étiquettes, ou s'il fallait un pointeur sur pointeur sur etiq. Pour des raisons de facilité, on a opté pour la deuxième option. Déceler une

étiquette était simple. Il suffisait à nouveau de vérifier la présence des éventuelles initialisations d'étiquettes, facilement repérable par la présence du caractère ' :'. On stock ensuite ces étiquettes dans notre tableau. La dernière opération sert à vérifier que le programme n'initialise pas la même étiquette à deux endroits. Nous avons donc parcouru notre tableau d'étiquettes par une double boucle, comparant chaque nom d'étiquette avec les autres. Ceci prend un temps θ(n^2), avec n le nombre d'étiquettes, mais est nécessaire.

Ensuite vient la boucle principale, qui devait parcourir chaque ligne du fichier et la traduire lorsque la syntaxe de celle-ci ne pose pas de problème grâce à une fonction traduction, dont je vous expliquerai l'implémentation plus bas. Contrairement à notre première idée de stocker nos lignes de codes assembleur dans un tableau ou une liste chainée, nous avons décidé de traduire directement chaque ligne récupérée dans notre fichier, et de l'insérer directement traduite dans notre nouveau fichier hexa.pgm. Ceci permettait une meilleure gestion de la mémoire, et le code nous semblait plus propre et efficace. Grace au numéro de ligne recueilli par le premier parcours du fichier source, on savait si oui ou non la ligne finissait par un '\n' (toutes sauf la dernière). Cependant, nous avons rencontré un problème dont nous n'avons trouvé la solution. En compilant notre programme sur nos ordinateurs personnels, la dernière ligne semblait en effet pas contenir un '\n' comme dernier caractère. Pourtant, sur les ordinateurs de Dauphine (en B0…), cette dernière contenait elle aussi ce caractère '\n'. Nous avons donc mis la condition qui enlève le dernier caractère d'une ligne tant que celle-ci n'est pas la dernière en commentaire, pour qu'elle n'agisse pas, pensant que vous essayerez notre programme sur les ordinateurs de dauphine. Ceci se trouve à la ligne 218 - *// if (cpt !=nligne) ;* -Il ne faut pas hésiter à l'enlever comme commentaire si le programme ne marche pas correctement. Nous nous excusons pour la gêne occasionnée.
On traduit ensuite notre fichier, ligne par ligne, en renvoyant une phrase d'erreur lorsqu'une des lignes comporte un code erroné. Nous aurions pu améliorer notre code en fournissant également le type d'erreur, ainsi qu'indiquer quel était le mot/donnée qui posait problème.
Enfin, par la même logique que précédemment, on ajoute un '\n' à la fin de la ligne exceptée la dernière, avant de l'insérer dans notre fichier hexa.pgm.

Finalement, on n'oublie pas de bien libérer la mémoire allouée, notamment pour le tableau d'etiq. Nous avons découvert plus tard que la fonction strdup allouait par elle-même un espace mémoire (ce qui nous semblait logique), et avons donc bien pensé à libérer la mémoire allouée.

traduction (la fonction):
On n'a pas eu beaucoup de mal à implémenter la fonction traduction.
Cependant, on a encore commis une faute dans un premier temps, qui était de penser que les instructions devaient bien être alignées les unes par rapport aux autres. On a réalisé après coup qu'on avait mal compris la syntaxe comme expliquée dans l'énoncé, parce qu'on s'était plutôt contenté de suivre l'exemple donné. En effet, les instructions étant bien alignées les unes par rapport aux autres dans l'exemple, et on avait donc créé une variable debut_instru de type int qui signalait après combien de caractères l'instruction devait commencer (au même endroit que commençait l'instruction qui suivait l'étiquette la plus longue).
Toutes erreurs corrigées, nous avons d'abord créer deux cas : si la ligne contient une étiquette ou non, qu'on vérifiait par la présence de ':' dans chaque ligne. On initialise ensuite une variable locale i, qui allait parcourir toute la ligne d'instruction en assembleur, afin de récupérer les instructions/données dans des variables, et ensuite les traduire en suivant la méthode suivante.
On parcourt d'abord les premiers caractères jusqu'au premier caractère de notre instruction après les étiquettes/espaces/tabulations. Arrivé à celui-ci on copie notre instruction dans une variable chaine de caractères instru. Grace à un tableau à 15 entrées, qui représentent les 15 instructions possibles, on compare l'instruction récupérée parmi celles qui existent. Si aucune comparaison n'aboutit, il y a forcément une erreur de syntaxe dans cette ligne, dans quel cas on renvoie « erreur ».
On continue à parcourir notre fichier, en distinguant les données attendues selon l'instruction (rien, une valeur ou une étiquette). On a découvert deux fonctions très intéressantes. La première, sprintf, traduit

un nombre en chaine de caractère sous forme hexadécimale, ce qui est très utile pour transformer notre valeur récupérée, et la renvoyer traduite. La deuxième, atoi, prend une chaine de caractères, et traduit celle-ci en un int, ce qui était particulièrement utile pour récupérer les données.
Les différents cas d'instructions étaient assez intéressants à gérer, selon si aucune données, une donnée, ou bien une étiquette suivait l'instruction. Nous avons réalisé à ce moment à quel point la manière dont nous avons gérer les étiquettes était pratique.

Exécution :

L'exécution du code s'avérait être bien plus facile que nous l'avions pensé. Nous avons d'abord eu un peu de mal à comprendre exactement comment fonctionne la machine à pile, à savoir si la mémoire était partagée ou non par nos registres PC et SP. En effet, les instructions de PC sont sur 5 octets, et une adresse mémoire pouvait en contenir seulement 4. En relisant attentivement le sujet, et en demandant un peu d'aide à des camarades de classe, nous avons finalement compris que la mémoire concernait seulement la pile.

De la même manière que pour les étiquettes, nous avons décidé de créer une structure pour les lignes d'instructions, qui contient deux paramètres : instru indique laquelle des 15 instructions on appelle, grâce au premier octet, et donnee la donnée transmise, qu'on peut lire sur les quatre derniers octets. On a ensuite initialisé un tableau d'instructions, et stocker chaque ligne d'instruction dedans. PC commençait bien à 0, et s'incrémentait à chaque fin d'instruction.

On a ensuite énuméré les 29 instructions possibles (avec les 15 de op), tout en vérifiant qu'il n'y ait pas d'erreurs de bornes de SP, qui ne doit jamais pointer sur des cases hors de [0,5000]. De plus, à chaque accès de mémoire, on devait vérifier que l'adresse utilisée se trouve bien entre 0 et 5000. En cas d'erreur, le programme s'arrête, et renvoie la phrase « Erreur de segmentation (core dumped). », petit clin d'œil à cette même phrase qu'on espérait ne jamais voir lors de la compilation de nos algorithmes.

La boucle doit s'arrêter lors d'un halt, dans quel cas on n'oublie pas de free notre tableau d'instruction. Par contre, notre adresse PC pouvait elle pointer sur autre chose qu'une instruction, dans quel cas on a renvoyé une phrase d'erreur à l'utilisateur, indiquant qu'on était arrivé à la fin de notre programme, sans jamais rencontrer d'instruction halt.

Test :
Pour tester notre programme, nous avons évidemment utilisé l'exemple, ainsi que des programmes que nous avons écrits, dans le but de passer par toutes les fonctions disponibles. Grace à ceux-ci, nous avons découvert de nombreux détails auxquels nous n'avions pas pensés. Parmi ceux-ci, on a trouvé :
-la division par 0 qui est impossible.
-la mauvaise implémentation de la fonction rnd, que nous avons corrigé grâce à la bibliothèque time, qui donne une nouvelle valeur à chaque nouvelle exécution du même programme.
-des oublis de conditions de bornes pour certaines fonctions, qui accédaient à des cases mémoires inexistantes.

Pour conclure, nous avons bien apprécié travailler ce sujet, qui nous offrait beaucoup de liberté et d'interprétation. Réfléchir plus largement aux différentes méthodes qui permettent de résoudre une tâche relève selon nous du métier de programmeur. En effet, on imagine que le programmeur est rarement amené à suivre des instructions précises de codes, et est plus amené à trouver par lui-même comment optimiser une réponse à un problème donné. Malheureusement, nous avons eu du mal à bien libérer totalement l'espace mémoire, faute de gestion de temps.